Application of K-Nearest Neighbors to Fraudulent Credit Card Transaction Classification

1 Introduction

The K-Nearest Neighbors (K-NN) model stands out for its simplicity and effectiveness among the algorithms we used to classify the fraudulent credit card transactions dataset. This section of the report examines the application of the K-NN model to the dataset, covering data preparation, feature selection, model training, and evaluation. By the end of this section, readers will gain a comprehensive understanding of how the K-NN algorithm was implemented in our project, the challenges encountered, and the outcomes achieved.

Despite its simplicity, the K-NN algorithm can achieve high performance when properly implemented and applied to the right context. For classification tasks, the algorithm assigns class labels by determining which data points in the training set are most similar to each data point in the test set. The class label is chosen from the k most similar "neighbors" through a majority vote. In the context of K-NN, similarity is measured using a distance metric, with common metrics including Euclidean, Manhattan, and Minkowski distances.

In this section, we will explore the steps involved in implementing the K-NN model for our dataset. We will start with data preparation, ensuring the data is clean and normalized. Next, we will discuss feature selection to identify the most relevant features for classification. The model training phase will cover the selection of the optimal k value, followed by an evaluation of the model's performance using cross-validation. Finally, we will reflect on the results obtained, describing key insights and any challenges we faced during the implementation process.

This report examines each phase of the K-NN model application to provide a clear and detailed account of how we used the algorithm to detect fraudulent transactions.

2 Data Preprocessing

Data preprocessing is crucial for any machine learning task, particularly for K-NN models, which are sensitive to feature scales. For this project, we used Scikit-learn's `RobustScaler`, which scales data using the interquartile range, minimizing the influence of outliers. We split the dataset into training (80%)

and test (20%) sets, fitting the scaler to the training set and transforming both sets accordingly.

```
from sklearn.preprocessing import RobustScaler
from sklearn.model_selection import train_test_split

# Splitting the data
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=
    y, test_size=0.2, random_state=42)

# Scaling the data
scaler = RobustScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)
```

Listing 1: Data Preprocessing

3 Feature Selection

We excluded features with a correlation close to 0 with the target variable. Specifically, any feature with an absolute correlation value less than 0.1 was removed from the dataset.

```
target_correlations = df.corr()['Class'].sort_values(ascending=
    False)
selected_features = target_correlations[target_correlations.abs()
    >= 0.1].index.tolist()
df_clean = df[selected_features]
```

Listing 2: Feature Selection

4 Model Implementation

We implemented the K-NN algorithm using a custom classifier class that includes methods for fitting the model to training data, predicting labels for test data, and evaluating performance. Euclidean distance was used as the distance metric, and we tested various values of k to identify the optimal number of neighbors. To evaluate performance, we performed 5-fold cross-validation using scikit-learn's `StratifiedKFold`. We averaged the accuracy, precision, and recall for each k value and recorded the results in a dictionary.

```
from numba import njit, prange

@njit(parallel=True, fastmath=True)
def euclidean_distance(X_train, X_test):
    num_train, num_features = X_train.shape
    num_test = X_test.shape[0]
    distances = np.empty((num_test, num_train), dtype=np.float64)

    for i in prange(num_test):
        for j in range(num_train):
            diff = X_train[j] - X_test[i]
```

```
            distances[i, j] = np.sqrt(np.sum(diff ** 2))

    return distances

@njit(parallel=True, fastmath=True)
def predict_labels(distances, y_train, k):
    num_test = distances.shape[0]
    predictions = np.empty(num_test, dtype=np.int32)

    for i in prange(num_test):
        neighbors_indices = np.argsort(distances[i])[:k]
        neighbor_labels = y_train[neighbors_indices]
        count_1 = np.sum(neighbor_labels == 1)
        count_0 = np.sum(neighbor_labels == 0)
        predictions[i] = 1 if count_1 > count_0 else 0

    return predictions

class KNNClassifier:

    def __init__(self, k=5):
        self.k = k

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train

    def predict(self, X_test, batch_size=100):
        num_samples = X_test.shape[0]
        predictions = np.empty(num_samples, dtype=np.int32)

        for i in range(0, num_samples, batch_size):
            end_index = min(i + batch_size, num_samples)
            batch_X_test = X_test[i:end_index]
            distances = euclidean_distance(self.X_train,
    batch_X_test)
            batch_predictions = predict_labels(distances, self.
    y_train, self.k)
            predictions[i:end_index] = batch_predictions

        return predictions

    def evaluate(self, X_test, y_test):
        self._predictions = self.predict(X_test)

        self._accuracy = np.sum(self._predictions == y_test) / len(
    y_test)

        # Compute the number of true positives, false positives,
    and false negatives
        true_positives = np.sum((self._predictions == 1) & (y_test
    == 1))
        false_positives = np.sum((self._predictions == 1) & (y_test
     == 0))
        false_negatives = np.sum((self._predictions == 0) & (y_test
     == 1))

```

```
        # Compute precision and recall
        self._precision = true_positives / (true_positives +
    false_positives) if (true_positives + false_positives) > 0 else
     0
        self._recall = true_positives / (true_positives +
    false_negatives) if (true_positives + false_negatives) > 0 else
     0

    def predictions(self):
        return self._predictions  # Return the predictions

    def metrics(self):
        return np.array([self._accuracy, self._precision, self.
    _recall])  # Return the metrics as an array

def cross_validate_knn(X, y, k_values, n_splits=5):
    kf = StratifiedKFold(n_splits=n_splits, shuffle=True,
    random_state=42)
    results = []

    for k in k_values:
        knn = KNNClassifier(k=k)
        accuracies = np.array([])
        precisions = np.array([])
        recalls = np.array([])

        for train_index, val_index in kf.split(X,y):
            X_train, X_val = X[train_index], X[val_index]
            y_train, y_val = y[train_index], y[val_index]

            knn.fit(X_train, y_train)
            knn.evaluate(X_val, y_val)
            accuracy, precision, recall = knn.metrics()

            accuracies=np.append(accuracies,accuracy)
            precisions=np.append(precisions,precision)
            recalls=np.append(recalls,recall)


        results=np.append(results,{
            'k': k,
            'accuracy': np.mean(accuracies),
            'precision': np.mean(precisions),
            'recall': np.mean(recalls)
        })


 print({
            'k': k,
            'accuracy': np.mean(accuracies),
            'precision': np.mean(precisions),
            'recall': np.mean(recalls)
        })

    return results
```

Listing 3: Model Implementation

5 Model Evaluation and Results

We tested k values ranging from 5 to 25 in increments of 5. Although each k value performed similarly during hyperparameter testing, k=5 yielded the best results. Ultimately, our model with k=5 achieved excellent performance on the test set, with 99.95% accuracy, 91.76% precision, and 79.59% recall.

```
results = cross_validate_knn(X_train_s, y_train, np.arange(5, 30, 5))

# Example result
{'k': 5, 'accuracy': 0.9995479382913823, 'precision': 0.91764706, 'recall': 0.79591837}
```

Listing 4: Model Evaluation

6 Analysis and Discussion

6.1 Feature Selection

The feature selection process of removing features with an absolute correlation of less than 0.1 with the target improved the performance by ignoring features that have a weak relationship with the target variable but have a significant effect on the distance between points. While this primitive technique was useful in the context of our problem, future analysis could benefit from more sophisticated feature selection methods to achieve even better results.

6.2 Robustness and Generalization

We used 5 folds in our cross-validation to ensure that the model's performance could be generalized across different subsets of the data set. Because each subset of the training set performed consistently well, we can be confident the model generalizes to unseen data.

6.3 Model Performance

While the model's high accuracy shows that K-NN can correctly classify transactions most of the time, it certainly does not paint a complete picture. A credit card transaction classification model must have a high precision rate to minimize the number of legitimate transactions flagged as fraudulent, but the most important metric for evaluating the model's performance in this problem is the recall, the proportion of the fraudulent transactions the algorithm classified as fraudulent. In an ideal situation, we like to design a model that classifies every fraudulent transaction as fraudulent, and while a 79.59% recall rate is decent, it shows that this model still has room to improve.

6.4 Limitations and Disadvantages of K-NN

Computational Efficiency: Calculating the distance between points and predicting labels are considerable bottlenecks in evaluating the test set's performance. To help mitigate this, we wrote JIT-compiled functions to increase the efficiency of calculating the distance between points and predicting class labels, but despite significant efforts, K-NN is still computationally expensive for evaluating large data sets. Future analysis could benefit from dividing the training set into regions, so data points in the test set only have to compute the distance of data points that appear in the same region.

Imbalanced Data Set: While we used stratification to ensure similar proportions of fraudulent transactions in different subsets of the data, fraudulent credit card transactions account for less than a fifth of a percent of all points in the data set. Because of the imbalance between classes, the legitimate transactions tend to dominate the voting process for higher values of k leading to lower recall rates.

6.5 Strengths of K-NN

Simplicity: While optimization for computational efficiency is quite difficult, K-NN models are simple to write. There aren't many hyperparameters to test, so hyperparameter tuning doesn't require testing as many combinations.

Interpretability: It's very easy to comprehend how a K-NN model classifies points in the test set.

Incremental Learning: It's easy to add and remove points from the model. In the context of credit card fraud, individuals make transactions every second of the day. When the goal is constantly updating the model with recent transactions, K-NN can easily implement these changes.